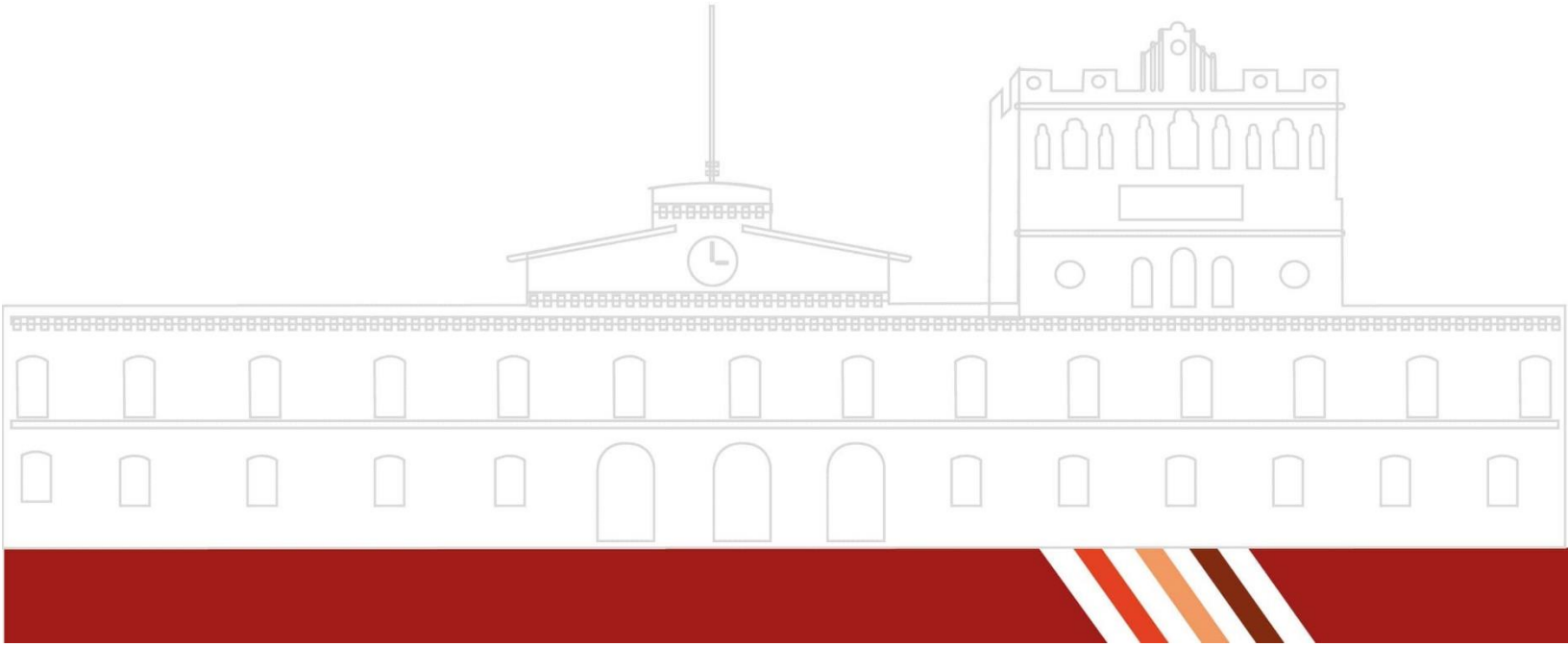
ANALISIS SEMÁNTICO

ALUMNOS:

José Manuel Hernández Lara

Joshua Pérez Reyes

Erick Meneses Melo



¿Qué es el análisis semántico?

La fase de análisis semántico de un procesador de lenguaje es aquélla que computa la información adicional necesaria para el procesamiento de un lenguaje, una vez que la estructura sintáctica de un programa haya sido obtenida. Es por tanto la fase posterior a la de análisis sintáctico y la última dentro del proceso de síntesis de un lenguaje de programación [Aho90]. Sintaxis de un lenguaje de programación es el conjunto de reglas formales que especifican la estructura de los programas pertenecientes a dicho lenguaje. Semántica de un lenguaje de programación es el conjunto de reglas que especifican el significado de cualquier sentencia sintácticamente válida. Finalmente, el análisis semántico1 de un procesador de lenguaje es la fase encargada de detectar la validez semántica de las sentencias aceptadas por el analizador sintáctico.

La fase de análisis semántico obtiene su nombre por requerir información relativa al significado del lenguaje, que está fuera del alcance de la representatividad de las gramáticas libres de contexto y los principales algoritmos existentes de análisis; es por ello por lo que se dice que captura la parte de la fase de análisis considerada fuera del ámbito de la sintaxis. Dentro del la clasificación jerárquica que Chomsky dio de los lenguajes, la utilización de gramáticas sensibles al contexto (o de tipo 1) permitirían identificar sintácticamente características como que la utilización de una variable en el lenguaje Pascal ha de estar previamente declarada.

Sin embargo, la implementación de un analizador sintáctico basado en una gramática de estas características sería computacionalmente más compleja que un autómata de pila. Así, la mayoría de los compiladores utilizan una gramática libre de contexto para describir la sintaxis del lenguaje y una fase de análisis semántico posterior para restringir las sentencias que "semánticamente" no pertenecen al lenguaje. En el caso que mencionábamos del empleo de una variable en Pascal que necesariamente haya tenido que ser declarada, el analizador sintáctico se limita a comprobar, mediante una gramática libre de contexto, que un identificador forma parte de una expresión.

Una vez comprobado que la sentencia es sintácticamente correcta, el analizador semántico deberá verificar que el identificador empleado como parte de una expresión haya sido declarado previamente. Para llevar a cabo esta tarea, es típica la utilización de una estructura de datos adicional denominada tabla de símbolos. Ésta poseerá una entrada por cada identificador declarado en el contexto que se esté analizando. Con este tipo de estructuras de datos adicionales, los desarrolladores de compiladores acostumbran a suplir las carencias de las gramáticas libres de contexto. Otro caso que se da en la implementación real de compiladores es ubicar determinadas comprobaciones en el analizador semántico, aun cuando puedan ser llevadas a cabo por el analizador sintáctico. Es factible describir una gramática libre de contexto capaz de representar que toda implementación de una función tenga al menos una sentencia return.

Sin embargo, la gramática sería realmente compleja y su tratamiento en la fase de análisis sintáctico sería demasiado complicada. Así, es más sencillo transferir dicha responsabilidad al analizador semántico que sólo deberá contabilizar el número de sentencias return aparecidas en la implementación de una función. El objetivo principal del analizador semántico de un procesador de lenguaje es asegurarse de que el programa analizado satisfaga las reglas requeridas por la especificación del lenguaje, para garantizar su correcta ejecución. El tipo y dimensión de análisis semántico requerido varía enormemente de un lenguaje a otro. En lenguajes interpretados como Lisp o Smalltalk casi no se lleva a cabo análisis semántico previo a su ejecución, mientras que, en lenguajes como Ada, el analizador semántico deberá comprobar numerosas reglas que un programa fuente está obligado a satisfacer.

Cómo el análisis semántico de un procesador de lenguaje no modela la semántica o comportamiento de los distintos programas construidos en el lenguaje de programación, sino que, haciendo uso de información parcial de su comportamiento, realiza todas las comprobaciones necesarias –no llevadas a cabo por el analizador para asegurarse de que el programa pertenece al lenguaje. Otra fase del compilador donde se hace uso parcial de la semántica del lenguaje es en la optimización de código, en la que analizando el significado de los programas previamente a su ejecución, se pueden llevar a cabo transformaciones en los mismos para ganar en eficiencia.

Existen dos formas de describir la semántica de un lenguaje de programación: mediante especificación informal o natural y formal.

La descripción informal de un lenguaje de programación es llevada a cabo mediante el lenguaje natural. Esto hace que la especificación sea inteligible (en principio) para cualquier persona. La experiencia nos dice que es una tarea muy compleja, si no imposible, el describir todas las características de un lenguaje de programación de un modo preciso. Como caso particular, véase la especificación del lenguaje ISO/ANSI C++ [ANSIC++].

La descripción formal de la semántica de lenguajes de programación es la descripción rigurosa del significado o comportamiento de programas, lenguajes de programación, máquinas abstractas o incluso cualquier dispositivo hardware. La necesidad de hacer especificaciones formales de semántica surge para [Nielson92, Watt96, Labra03]:

- Revelar posibles ambigüedades existentes implementaciones de procesadores de lenguajes o en documentos descriptivos de lenguajes de programación.
- Ser utilizados como base para la implementación de procesadores de lenguaje.
- Verificar propiedades de programas en relación con pruebas de corrección o información relacionada con su ejecución.
- Diseñar nuevos lenguajes de programación, permitiendo registrar decisiones sobre construcciones particulares del lenguaje, así como permitir descubrir posibles irregularidades u omisiones.
- Facilitar la comprensión de los lenguajes por parte del programador y como mecanismo de comunicación entre diseñador del lenguaje, implementador y programador. La especificación semántica de un lenguaje, como documento de referencia, aclara el comportamiento del lenguaje y sus diversas construcciones.
- Estandarizar lenguajes mediante la publicación de su semántica de un modo no ambiguo. Los programas deben poder procesarse en otra implementación de procesador del mismo lenguaje exhibiendo el mismo comportamiento.

El análisis semántico de un procesador de lenguaje es la fase encargada de detectar la validez semántica de las sentencias aceptadas por el analizador sintáctico.

Comprobaciones pospuestas por el analizador sintáctico:

A la hora de implementar un procesador de un lenguaje de programación, es común encontrarse con situaciones en las que una gramática libre de contexto puede representar sintácticamente propiedades del lenguaje; sin embargo, la gramática resultante es compleja y difícil de procesar en la fase de análisis sintáctico. En estos casos es común ver cómo el desarrollador del compilador escribe una gramática más sencilla que no representa detalles del lenguaje, aceptándolos como válidos cuando realmente no pertenecen al lenguaje. En la posterior fase de análisis semántico será donde se comprueben aquellas propiedades del lenguaje que, por sencillez, no fueron verificadas por el analizador sintáctico.

Hay multitud de escenarios de ejemplo y están en función de la implementación de cada procesador. Sin embargo, los siguientes suelen ser comunes:

– Es posible escribir una gramática libre de contexto capaz de representar que toda implementación de una función en C tenga al menos una sentencia return. No obstante, si escribimos la gramática de cualquier función como una repetición de sentencias, siendo return es un tipo de sentencia, la gramática es más sencilla y fácil de procesar. El analizador semántico deberá comprobar, pues, dicha restricción.

– Cuando un lenguaje posee el operador de asignación como una expresión y no como una sentencia (C y Java frente a Pascal), hay que comprobar que la expresión de la parte

izquierda de la asignación posee una dirección de memoria en la que se pueda escribir (lvalue). Esta restricción puede ser comprobada por el analizador semántico, permitiendo sintácticamente que cualquier expresión se encuentre en la parte izquierda del operador de asignación.

– Las sentencias break y continue de Java y C sólo pueden utilizarse en determinadas estructuras de control del lenguaje. Éste es otro escenario para que el analizador sintáctico posponga la comprobación hasta la fase análisis semántico.

Comprobaciones dinámicas:

Todas las comprobaciones semánticas descritas en este punto suelen llevarse a cabo en fase de compilación y por ello reciben el nombre de "estáticas". Existen comprobaciones que, en su caso más general, sólo pueden ser llevadas a cabo en tiempo de ejecución y por ello se llaman "dinámicas". Éstas suelen ser comprobadas por un intérprete o por código de comprobación generado por el compilador –también puede darse el caso de que no se comprueben. Diversos ejemplos pueden ser acceso a un vector fuera de rango, utilización de un puntero nulo o división por cero.

Comprobaciones de tipo:

Sin duda, este tipo de comprobaciones es el más exhaustivo y amplio en fase de análisis semántico. Ya bien sea de un modo estático (en tiempo de compilación), dinámico (en tiempo de ejecución) o en ambos, las comprobaciones de tipo son necesarias en todo lenguaje de alto nivel. De un modo somero, el analizador semántico deberá llevar a cabo las dos siguientes tareas relacionadas con los tipos:

1. Comprobar las operaciones que se pueden aplicar a cada construcción del lenguaje. Dado un elemento del lenguaje, su tipo identifica las operaciones que sobre él se pueden aplicar. Por ejemplo, en el lenguaje Java el operador de producto no es aplicable a una referencia a un objeto. De un modo contrario, el operador punto sí es válido.

2. Inferir el tipo de cada construcción del lenguaje. Para poder implementar la comprobación anterior, es necesario conocer el tipo de toda construcción sintácticamente válida del lenguaje. Así, el analizador semántico deberá aplicar las distintas reglas de inferencia de tipos descritas en la especificación del lenguaje de programación, para conocer el tipo de cada construcción del lenguaje.

Análisis Semántico como Decoración del AST:

El análisis semántico de un programa es más sencillo de implementar si se emplean para las fases de análisis sintáctico y semántico dos o más pasadas. En este caso, la fase de análisis sintáctico creará un árbol sintáctico abstracto para que sea procesado por el analizador semántico. Si el procesador es de una pasada, el analizador sintáctico irá llamando al analizador semántico de un modo recursivo y, si bien ningún árbol es creado de forma explícita, los ámbitos de las invocaciones recursivas (o los niveles de la pila del reconocedor) formarán implícitamente el árbol sintáctico.

Árbol de sintaxis abstracta

Como sabemos, un árbol sintáctico17es una representación de la estructura de una consecución de componentes léxicos (tokens), en la que éstos aparecen como nodos hoja y los nodos internos representan los pasos en las derivaciones de la gramática asociada.

Los árboles sintácticos poseen mucha más información de la necesaria para el resto de las fases de un compilador, una vez finalizada la fase de análisis sintáctico. Una simplificación de árbol sintáctico que represente toda la información necesaria para el resto del procesamiento del programa de un modo más eficiente que el árbol sintáctico original, recibe el nombre de árbol de sintaxis abstracta (AST, Abstract Syntax Tree).

Así, la salida generada por un analizador sintáctico de varias pasadas, será el AST representativo del programa de entrada. Un AST puede ser visto como el árbol sintáctico de una gramática denominada abstracta, al igual que un árbol sintáctico es la representación de una gramática (en ocasiones denominada concreta).

Por tanto, es común ver una gramática que representa una simplificación de un lenguaje de programación denominada gramática abstracta del lenguaje.

Ejemplo

Lo siguiente es una gramática (concreta) de una expresión, descrita en la notación propia de yacc/bison [Mason92]:

```
expresion: expresion '+' termino
         | expresion '-' termino
         | termino
         ;
termino: termino '*' factor
       | termino '/' factor
       | factor
       ;
factor: '-' factor
       | '(' expresion ')'
       | CTE_ENTERA
       ;
```

La sentencia 3*(21+-32)pertenece sintácticamente al lenguaje y su árbol sintáctico es:

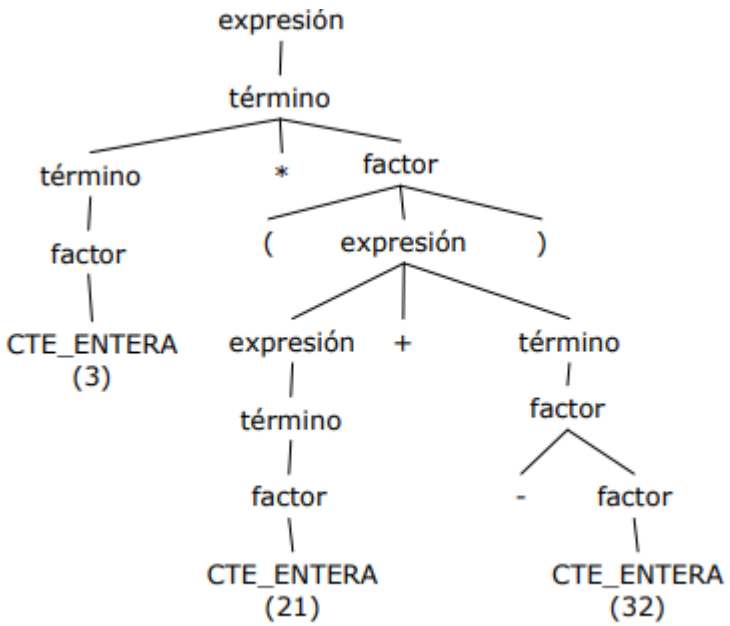


Figura 2: Árbol sintáctico generado para la sentencia 3*(21+-32).

En el árbol anterior hay información (como los paréntesis o los nodos intermedios factor y término) que no es necesaria para el resto de fases del compilador. La utilización de un AST simplificaría el árbol anterior. Una posible implementación sería siguiendo el siguiente diagrama de clases:

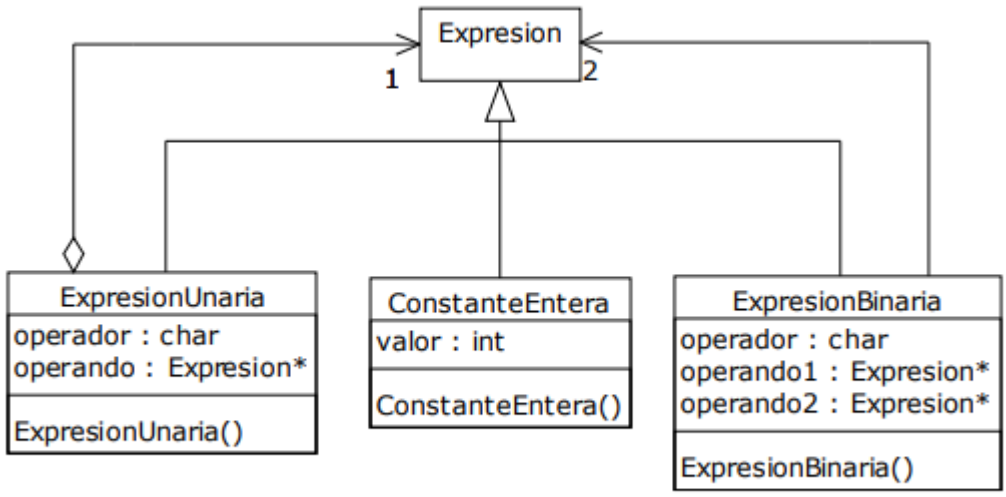


Figura 3: Diagrama de clases de los nodos de un AST.

Su implementación en C++ puede consultarse en el apéndice A.1. Todos los nodos son instancias derivadas de la clase abstracta expresión. De este modo, se podrá trabajar con cualquier expresión, independientemente de su aridad, mediante el uso de esta clase. Todas las expresiones binarias serán instancias de la clase ExpresionBinaria, teniendo un atributo que denote su operador. Del mismo modo, las expresiones con un operador unario (en nuestro ejemplo sólo existe el menos unario) son instancias de ExpresionUnaria. Finalmente, las constantes enteras son modeladas con

la clase ConstanteEntera. Una implementación yacc/bison que cree el AST a partir de un programa de entrada es:

```
%{
#include "ast.h"
int yyparse();
int yylex();
%}
%union {
  int entero;
  Expresion *expresion;
  }
%token <entero> CTE_ENTERA
%type <expresion> expresion termino factor
%%
expresion: expresion '+' termino {$$=new ExpresionBinaria('+',$1,$3); }
       | expresion '-' termino {$$=new ExpresionBinaria('-',$1,$3); }
       | termino {$$=$1;}
       ;
termino: termino '*' factor {$$=new ExpresionBinaria('*',$1,$3); }
       | termino '/' factor {$$=new ExpresionBinaria('/',$1,$3); }
       | factor { $$=$1; }
       ;
factor: '-' factor { $$=new ExpresionUnaria('-',$2); }
       | '(' expresion ')' { $$=$2; }
       | CTE_ENTERA { $$=new ConstanteEntera($1); }
       ;
%%
```

Así, ante la misma sentencia de entrada 3*(21+-32), el AST generado tendrá la siguiente estructura:

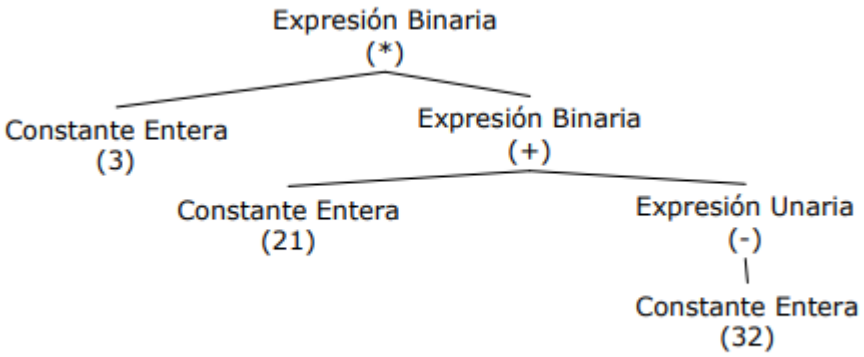


Figura 4: AST generado para la sentencia 3*(21+-32).

Nótese cómo ya no son necesarios los nodos de factor y término, puesto que el propio árbol ya se creará con una estructura que refleje la información relativa a la precedencia de operadores. De la misma forma, el procesamiento del AST en las siguientes fases es notablemente más sencillo que el del árbol sintáctico original. La gramática abstracta de nuestro ejemplo es:

```
expresion: expresionBinaria
         | expresionUnaria
         | constanteEntera
         ;
expresioBinaria: expresion ('*'|'/'|'+'|'-') expresion
         ;
expresionUnaria: '-' expresion
         ;
constanteEntera: CTE_ENTERA
         ;
```

Ejemplo

Considérese la siguiente gramática libre de contexto, que representa una simplificación de la sintaxis de una sentencia condicional en un lenguaje de programación imperativo –los símbolos terminales se diferencian de los no terminales porque los primeros han sido escritos en negrita:

```
sentencia     →  sentenciaIf
              |  lectura
              |  escritura
              |  expresion
sentenciaIf   →  if ( expresion ) sentencia else
else          →  else sentencia
              |  λ
lectura       →  read expresion
escritura     →  write expresion
expresion     →  true
              |  false
              |  cte_entera
              |  id
              |  expresion + expresion
              |  expresion - expresion
              |  expresion * expresion
              |  expresion / expresion
              |  expresion = expresion
```

La gramática anterior acepta más sentencias que las pertenecientes al lenguaje, como suele suceder en la mayor parte de los casos. El analizador semántico debería restringir la validez semántica de las sentencias teniendo en cuenta que la expresión de la estructura condicional debe ser lógica (o entera en el caso de C), y que tanto la expresión a la izquierda del operador de asignación como la expresión de la sentencia de lectura sean lvalues. La siguiente sentencia es sintácticamente válida:

```
if (true)
        read a
else
        write a=2+a*b
```

La implementación de un procesador del lenguaje presentado en la que se emplee más de una pasada, un AST apropiado generado por el analizador sintáctico podría ser el mostrado en la Figura 5. Nótese cómo el AST representa, de un modo más sencillo que el árbol sintáctico, la estructura de la sentencia condicional con tres nodos hijos: la expresión de la condición, la sentencia a ejecutar si la condición es válida, y la asociada al valor falso de la condición. Del mismo modo, las expresiones creadas poseen la estructura adecuada para ser evaluadas con un recorrido en profundidad infijo, de un modo acorde a las precedencias.

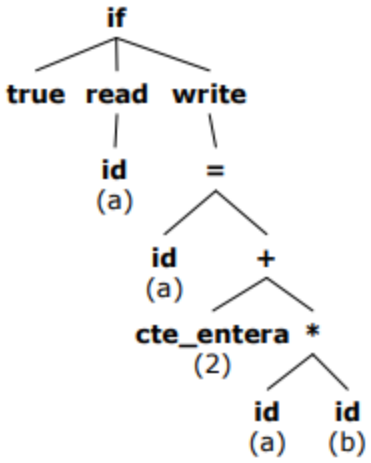


Figura 5: AST generado para la gramática y lenguaje de entrada presentados.

Decoración del AST

Ejemplo

Una vez que el analizador semántico obtenga el AST del analizador sintáctico, éste deberá utilizar el AST para llevar a cabo todas las comprobaciones necesarias para verificar la validez semántica del programa de entrada. Este proceso es realizado mediante lo que se conoce como decoración o anotación del AST: asignación de información adicional a los nodos del AST representando propiedades de las construcciones sintácticas del lenguaje, tales como el tipo de una expresión.

Un AST decorado o anotado18 es una ampliación del AST, en el que a cada nodo del mismo se le añaden atributos indicando las propiedades necesarias de la construcción sintáctica que representan.

Dada la gramática libre de contexto:

```
S            →   S declaracion ;
             |   S expresion ;
             |   λ
declaracion  →   int id
             |   float id
expresion    →   id
             |   cte_entera
             |   cte_real
             |   ( expresion )
             |   expresion + expresion
             |   expresion - expresion

             |   expresion * expresion
             |   expresion / expresion
             |   expresion = expresion
```

Para implementar un analizador semántico deberemos decorar el AST con la siguiente información:

− Las expresiones tendrán asociadas un atributo tipo que indique si son reales o enteras. Esto es necesario porque se podrá asignar un valor entero a una expresión real, pero no al revés.

− Las expresiones deberán tener un atributo lógico que indique si son o no lvalues. De este modo, se podrá comprobar si lo que está a la izquierda de la asignación es o no semánticamente correcto.

− En una declaración se deberá insertar el identificador en una tabla de símbolos con su tipo declarado, para poder conocer posteriormente el tipo de cualquier identificador en una expresión. Es, por tanto, necesario asignar un atributo nombre (cadena de caracteres) a un identificador.

− Finalmente –aunque más enfocado a la fase de generación de código o interpretación que al análisis semántico– se le asigna un valor entero o real a las constantes del lenguaje.

Para el siguiente programa, un posible AST decorado es el mostrado en la Figura 6.

```
int a;
float b;
b=(a+1)*(b-8.3);
```

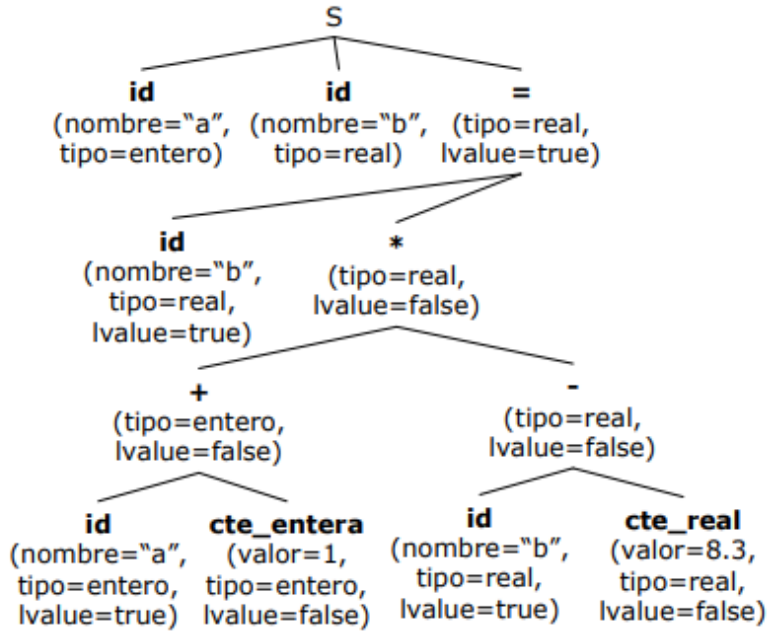


Figura 6: AST decorado para llevar a cabo el análisis semántico del programa analizado.

Nótese cómo el analizador semántico será el encargado de decorar el AST como se muestra en la Figura 6, además de comprobar las restricciones mencionadas con anterioridad – por las que, precisamente, se ha decorado el árbol.

De este modo es más sencillo llevar a cabo la tarea de análisis semántico puesto que toda la información sintáctica está explícita en el AST, y el análisis semántico tiene que limitarse a decorar el árbol y comprobar las reglas semánticas del lenguaje de programación. En el ejemplo anterior, utilizando la estructura del árbol se va infiriendo el tipo de cada una de las expresiones.

La información anotada a cada nodo del árbol será empleada también por las siguientes fases del procesador de lenguajes como la generación de código.

Siguiendo con el ejemplo, el conocer los tipos de cada una de las construcciones del AST facilita al generador de código saber el número de bytes que tiene que reservar, la instrucción de bajo nivel que tiene que emplear, o incluso si es necesario o no convertir los operandos antes de realizar la operación

El principal formalismo que existe para decorar árboles sintácticos es el concepto de gramática atribuida. Mediante gramáticas atribuidas se implementan analizadores semánticos a partir del AST o del árbol sintáctico.

También, partiendo de una gramática atribuida, existen métodos de traducción de éstas a código. Estos conceptos y técnicas serán lo que estudiaremos en los siguientes puntos.

Referencias

- Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman. Compiladores: Principios, Técnicas y Herramientas. Addison-Wesley Iberoamericana. 1990.
- R.G. Atkinson. Hurricane: An Optimizing Compiler for Smalltalk. ACM SIGPLAN Notices, vol. 21, no. 11. 1986.
- Kurt M. Bischoff. User Manual for Ox: an Attribute Grammar Compiling System based on Yacc, Lex and C. Technical Report 92-30. Department of Computer Science, Iowa State University. 1992.
- D. Bjorner, C.B. Jones. Formal Specification and Software Development. Prentice Hall. 1982.
- Luca Cardelli. Type Systems. The Computer Science and Engineering Handbook, CRC Press. 1997.
- J.M. Cueva, R. Izquierdo, A.A. Juan, M.C. Luengo, F. Ortín, J.E. Labra. Lengua jes, Gramáticas y Autómatas en Procesadores de Lenguaje. Servitec. 2003.
- Juan Manuel Cueva Lovelle. Conceptos básicos de Procesadores de Lenguaje. Servitec. 2003.
- Bruce Eckel. Thinking in C++. Volume 1: Introduction to Standard C++. 2nd Edition. Prentice Hall. 2000.
- J.E. Hopcroft, R. Motwani, J.D. Ulman. Introducción a la Teoría de autómatas, lenguajes y computación. Pearson Educación. 2002.